Searching for Machine Learning Pipelines Using a Context-Free Grammar

Radu Marinescu[1], **Akihiro Kishimoto**[1], **Parikshit Ram**[1], **Ambrish Rawat**[1], **Martin Wistuba**[1], **Paulito P. Palmes**[1], **Adi Botea**[2*]

[1] IBM Research
[2] Eaton

radu.marinescu@ie.ibm.com, Akihiro.Kishimoto@ibm.com, parikshit.ram@ibm.com, Ambrish.Rawat@ie.ibm.com, Martin.Wistuba@ibm.com, paulpalmes@ie.ibm.com, adi.botea@eaton.com

Abstract

AutoML automatically selects, composes and parameterizes machine learning algorithms into a workflow or pipeline of operations that aims at maximizing performance on a given dataset. Although current methods for AutoML achieved impressive results they mostly concentrate on optimizing fixed linear workflows. In this paper, we take a different approach and focus on generating and optimizing pipelines of complex directed acyclic graph shapes. These complex pipeline structure may lead to discovering new synthetic features and thus boost performance considerably. We explore the power of heuristic search and context-free grammars to search and optimize these kinds of pipelines. Experiments on various benchmark datasets show that our approach is highly competitive and often outperforms existing AutoML systems.

Introduction

Automated Machine Learning (or AutoML for short) seeks to automatically compose and parameterize machine learning algorithms to maximize a given metric such as predictive accuracy on a given dataset. It has received increasing attention over the past decades but it became even more acute in light of the recent explosion in machine learning applications. AutoML has gradually extended from hyper-parameter optimization (HPO) for the best configuration of a single machine learning algorithm (Bergstra et al. 2011; Snoek, Larochelle, and Adams 2012) to tackling the optimization of the entire machine learning pipeline from data preparation to model learning (Feurer et al. 2015). Several AutoML international challenges have been organized in the last decade (Guyon, Bennett, and Cawley 2015; Guyon and Tu 2018), spurring the development of efficient AutoML systems such as Auto-WEKA (Kotthoff et al. 2017), TPOT (Olson et al. 2016) and the challenge winner Auto-sklearn (Feurer et al. 2015).

In this paper, we focus on this problem of automatic configuration of machine learning pipelines. Existing AutoML systems can be divided into two main categories. First, the AutoML problem is designed as an optimization problem with a fixed number of decision variables, which then is solved in many different ways, including standard Bayesian optimization (Hutter, Hoos, and Leyton-Brown 2011), mixed integer-continuous non-linear programming (Liu et al. 2020), hierarchical task networks (Mohr, Wever, and Hullermeier 2018) or Monte-Carlo tree search coupled with Bayesian optimization (Rakotoarison, Schoenauer, and Sebag 2019). Specifically, these approaches assume a fixed linear structure of the pipeline and therefore have one variable for a preprocessing algorithm, one variable for the learning algorithm, and one variable for each parameter of each algorithm which leads to a solution space of fixed dimensionality. On the other hand, approaches of the second category attempt to relax the fixed structure of the pipeline. Consequently, these methods are able to generate more flexible ML pipelines that allow for complex preprocessing steps at the expense of exploring a much larger search space. More specifically, systems like TPOT (Olson et al. 2016), RECIPE (de Sa et al. 2017) and ML-PLAN-UL (Wever, Mohr, and Hullermeier 2018) organize the search space in terms of a formal grammar that naturally allows for recursive structures and employ evolutionary as well as planning techniques to explore the search space more effectively. However, as in the previous category, the use of the grammar is limited to chain-shaped (or at most tree-shaped) data-flow graphs.

Contribution: We consider the second category of AutoML systems and show how to exploit the power of heuristic search and context-free grammars to generate ML pipelines of complex Directed Acyclic Graph (DAG) shapes. Specifically, we propose a simple yet expressive context-free grammar that allows us to construct arbitrarily complex pipeline structures that extend well beyond the traditional preprocessing-estimation linear workflow. These complex pipelines allow naturally for stacking or concatenating the outputs of multiple transformers/estimators in a DAG structure and using them as new "synthetic features" for downstream estimators in the pipeline. This in turn may translate into improved performance. The proposed grammar defines the search space of all possible DAG-shaped pipelines. For a given number of transformers and estimators this search space can be many orders of magnitude larger than that corresponding to fixed linear pipelines. Therefore, we develop a greedy best-first search algorithm that traverses efficiently the space of partial pipelines guided by an effective heuristic function to find the most promising pipeline structure. Once the most promising pipeline structure is identified, we subsequently optimize it by selecting the best configuration

*Work done while at IBM Research.

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

of machine learning algorithms together with their hyper-parameters. This way we can leverage any of the existing joint optimizers developed for fixed linear pipelines. Therefore, we put together all these ideas into a new AutoML system called PIPER and evaluate it extensively on a variety of benchmark datasets. Our results show conclusively that PIPER is highly competitive and often outperforms existing state-of-the-art AutoML systems.

Related Work

Previous work on AutoML has focused on optimizing a fixed linear pipeline structure consisting of a data preparation step, a feature transformation step followed by the model learning step. Auto-WEKA (Thornton et al. 2012; Kotthoff et al. 2017) and Auto-sklearn (Feurer et al. 2015) are the main representatives for solving AutoML that apply the general purpose algorithm configuration framework SMAC (Hutter, Hoos, and Leyton-Brown 2011) based on Bayesian optimization (BO) to find optimal machine learning pipelines (Snoek, Larochelle, and Adams 2012; Hutter, Hoos, and Leyton-Brown 2011; Bergstra et al. 2011).

TPOT (Olson et al. 2016) is the first system that attempts to generate non-linear pipelines. It uses genetic programming to splice and concatenate randomly generated tree-shaped pipelines. One major drawback of TPOT is that it tends to generate a lot of invalid pipelines that cannot be trained. Therefore, more recent genetic programming based systems like RECIPE (de Sa et al. 2017), Auto-MEKA (de Sa, Freitas, and Pappa 2018) or Auto-DSGE (Assuncao et al. 2020) utilize a context-free grammar to generate and evolve only valid pipelines. However, none of these systems can generate and optimize complex DAG shaped pipelines.

ML-Plan (Mohr, Wever, and Hullermeier 2018) is a recent system that assumes a fixed linear structure of the pipelines and uses a form of AI planning called *hierarchical task networks* (HTN) to optimize the pipeline. In practice, it was highly competitive with Auto-sklearn and TPOT. The system was extended to deal with unlimited length pipelines thus allowing for more complex pre-processing workflows (Wever, Mohr, and Hullermeier 2018).

AlphaD3M (Drori et al. 2018) integrates reinforcement learning (RL) with Monte-Carlo Tree Search (MCTS) for solving AutoML problems but without imposing efficient decomposition over hyper-parameters and model selection. AutoStacker (Chen et al. 2018) focuses on ensembling and cascading to generate complex pipelines while the algorithm selection and hyper-parameter optimization is done using a discrete evolutionary search. Unlike our approach, AutoStacker does not consider transformers and therefore its search space is a subset of the search space defined by our grammar. Furthermore, our technique provides a natural decomposition of the search problem where heuristic search is used for the discrete optimization part and Bayesian optimization (or other relevant solvers) are used for the (continuous) hyper-parameter optimization part.

MOSAIC (Rakotoarison, Schoenauer, and Sebag 2019) combines MCTS with BO to optimize the algorithm selection and their hyper-parameters in a fixed pipeline structure. (Liu et al. 2020) developed an iterative ADMM based joint-optimization method that splits the algorithm selection phase and hyper-parameter optimization into two simpler subproblems using the augmented Lagrangian function. The approach is also restricted to fixed pipeline structures.

More recently, (Katz et al. 2020) use HTN planning and a context-free grammar to generate DAG-shaped pipelines. The system performs a problem agnostic "one-shot" enumerations of pipelines from the grammar which is shown to not perform significantly much better (or much worse) than a fixed 3-stage baseline. Furthermore, their problem dependent feedback scheme was shown to not perform better than the one-shot scheme. In contrast, our proposed system is problem adaptive and significantly outperforms the fixed 3-stage baseline (with the same joint optimizer) across various datasets as we will show in the experimental section.

Neural architecture search (NAS) is an area of AutoML which focuses on optimizing the architecture of a neural network. A variety of optimization methods has been explored for this task including RL (Zoph and Le 2017), evolutionary algorithms (Liu et al. 2018; Real et al. 2019), MCTS (Wang et al. 2019), and BO (Kandasamy et al. 2018). Similar to the problem of pipeline selection, a DAG containing different operators has to be generated. However, this problem is oftentimes reduced to finding a smaller DAG which is then repeated several times (Real et al. 2019). Furthermore, all state-of-the-art optimizers rely on key elements unique to NAS that do not trivially transfer to optimizing ML pipelines. Examples for these elements are weight-sharing between architectures (Pham et al. 2018) and a fully differentiable search that allows to learn the composition of the DAG (Liu, Simonyan, and Yang 2019). Therefore, NAS and pipeline selection for tabular data are related problems, which use similar techniques, but their actual execution is different.

Background

Context-Free Grammars

A *Context-Free Grammar* (CFG) (Segovia-Aguas, Jimenez, and Jonsson 2017) is a tuple $\mathcal{G} = \langle V, v_0, \Sigma, R \rangle$, where V is a finite set of non-terminal symbols, $v_0 \in V$ is the start non-terminal symbol, $\sigma \in \Sigma$ is a finite set of terminal symbols, and $R = \{\alpha \to \beta | \alpha \in V, \beta \in (V \cup \Sigma)^*\}$ is a finite set of production rules in the grammar. The semantics of the CFG is as follows: each $v \in V$ represents a sub-language of the language defined by the grammar and Σ is the alphabet of the language defined by $\mathcal{G}$ and can contain the empty string, which we denote by $\epsilon \in \Sigma$. We denote by e_0 the string that contains only the initial start symbol v_0.

Given a string $e_1 = u_1 \alpha u_2 \in (V \cup \Sigma)^*$ and a rule $r : \alpha \to \beta \in R$, we say that e_1 directly yields $e_2 = u_1 \beta u_2$, which we denote by $e_1 \to_r e_2$. Furthermore, we say that e_1 yields e_n (denoted by $e_1 \to^* e_n$ iff there exist strings $e_1, \ldots, e_n \in (V \cup \Sigma)^*$ and rules $r_1, \ldots, r_{n-1} \in R$, such that for all $1 \leq i < n$ we have that $e_i \to_{r_i} e_{i+1}$. In this case, we say that $r_1 \cdot \ldots \cdot r_{n-1}$ is an *inducing sequence of rules* for the pair of strings (e_1, e_n).

The language of a CFG, $L(\mathcal{G}) = \{e \in \Sigma^* : v_0 \to^* e\}$ is the set of all strings that contain only terminal symbols and

that can be produced from the start symbol v_0.

Heuristic Search

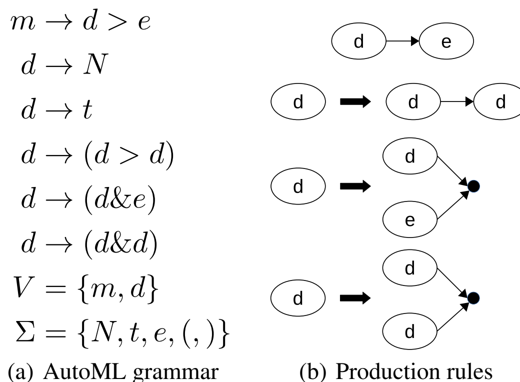
Consider a search space defined implicitly by a set of states (the nodes in the graph), operators that map states to states having costs or weights (the directed weighted arcs), a starting state n_0 and a set of goal states. The task is to find the least cost solution path from n_0 to a goal (Nilsson 1980), where the cost of a solution path is the sum of the weights or the product of the weights on its arcs. Best-First Search (BFS) is perhaps the most effective search algorithms to find the optimal solution path. Specifically, it explores the search space using a heuristic evaluation function $f(n)$ that estimates the best cost solution path that passes through each node n. It is known that when $f(n)$ is a lower bound on the optimal cost path the algorithm terminates with an optimal solution (Nilsson 1980; Pearl 1984).

The most popular variant of BFS is A* (P. Hart and Raphael 1968), whose heuristic evaluation function is $f(n) = g(n) + h(n)$, where $g(n)$ is minimal cost from the root n_0 to n along the current path, and $h(n)$ underestimates $h^*(n)$, the optimal cost from n to a goal node. A* search is known to be optimally efficient, namely it expands the smallest number of nodes compared with any other search strategy using the same heuristic function (Dechter and Pearl 1985).

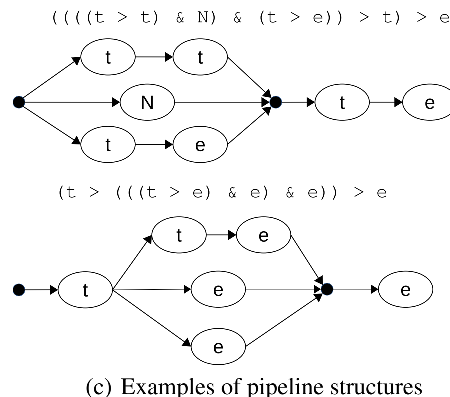
Greedy Best-First Search (GBFS) belongs to the family of best-first search algorithms and uses $f(n) = h(n)$. Unlike BFS, GBFS is not guaranteed to terminate with the optimal solution, however, in many cases (e.g, (Helmert 2006)) is able to find reasonably good quality solutions relatively quickly.

$$
\begin{aligned}
& m \rightarrow d > e \\
& d \rightarrow N \\
& d \rightarrow t \\
& d \rightarrow (d > d) \\
& d \rightarrow (d \& e) \\
& d \rightarrow (d \& d) \\
& V = \{m, d\} \\
& \Sigma = \{N, t, e, (,)\}
\end{aligned}
$$

(a) AutoML grammar



(b) Production rules



(c) Examples of pipeline structures

Figure 1: A context-free grammar for generating complex pipeline structures (or shapes).

A Grammar For Generating Machine Learning Pipelines

In this section, we introduce a context-free grammar that allows us to construct arbitrarily complex pipeline shapes that extend well beyond the standard preprocessing-estimation linear workflows. We will define complex pipeline structures using two combinators denoted by $>$ and $\&$ which resemble those defined in the LALE library (Hirzel et al. 2019).

The $>$ combinator performs the *pipe* operation, where $\alpha > \beta$ is a pipeline where the data goes into the α operator and the output of α is piped into the β operator. The equivalent, but more verbose, way of performing this in scikit-learn is `make_pipeline([`α, β`])`. The & combinator performs parallel independent executions, where α & β is a (partial) pipeline with the operators α and β applied to the data independently in parallel. The output of this (partial) pipeline is typically piped into a special purpose operator to concatenate (or horizontally stack) the features. Therefore, the pipeline would be defined as $(\alpha \& \beta) > \gamma$, where γ is the concatenation operator, however, for simplicity, we will not show the concatenation step explicitly in the formal definition the grammar. This can be equivalently performed in scikit-learn with `make_union([`α, β`])`. Note that, in the above examples, the operators α and β can be ML operators as well as (partial) pipelines.

AutoML Grammar We are now ready to present our context-free grammar for pipeline structures. Specifically, let $\mathcal{G}_{ML} = \langle V, v_0, \Sigma, R \rangle$ be the grammar shown in Figure 1(a) where the set of non-terminal and terminal symbols are $V = \{m, d\}$ and $\Sigma = \{N, t, e, (,)\}$, respectively and the start symbol is $v_0 = m$. For our purpose, the terminal symbols have the following semantics: N stands for a no-operation (NoOp) operator implying that the data is passed as is, e stands for an *estimator* (or a ML modeling operator such as a decision tree or a support vector machine), and t is used to denote a *transformer* (or a ML data-preprocessing and transformation operator such as normalization or PCA). The parenthesis (and) are used to group the operators in a consistent manner.

The set R contains 6 production rules that are used to encode the pipeline shape. The first rule for the non-terminal start symbol m indicates that the pipeline contains a data flow graph (denoted by the non-terminal d) piped into a ML modeling step (or estimator e) as depicted graphically at the top of Figure 1(b). The following rules correspond to the non-terminal symbol d and allow it to be: (i) a no-operation (NoOp), implying that data is passed as is, (ii) a t corresponding to ML data-preprocessing and transformation operators, (iii) a linear extension of the pipeline via the recursion $(d > d)$ thus allowing the pipeline to have arbitrary length and (iv) two other forms of extending the pipeline data-flow graph via the recursions $(d \& e)$ and $(d \& d)$ which

Algorithm 1 PIPER: greedy best-first search for pipeline structure generation and optimization

Require: Grammar $\mathcal{G}_{ML} = \langle V, v_0, \Sigma, R \rangle$, dataset $\mathcal{D}$
1: **procedure** GBFS
2: Let $pipeline = \emptyset$
3: Let v_0 be the start symbol of $\mathcal{G}$
4: Create a node $r \in S_{\mathcal{G}}$ and set $r.symbols = v_0$
5: Let $h(r)$ = ESTIMATE(r, $\mathcal{D}$), $OPEN = \{r\}$
6: **while** $OPEN$ is not empty or no timeout **do**
7: Select n from the top of $OPEN$
8: Remove n from $OPEN$
9: **if** $\forall s \in n.symbols, s \in \Sigma$ **then**
10: $pipeline = n.symbols$
11: **break**
12: $succ(n)$ = EXPAND(n)
13: **for all** $ch \in succ(n)$ **do**
14: **if** $ch \notin OPEN$ **then**
15: Let $h(ch)$ = ESTIMATE(ch, $\mathcal{D}$)
16: $OPEN = OPEN \cup \{ch\}$
17: Sort $OPEN$ in ascending order of $h(n)$
18: **return** OPTIMIZE($pipeline$, $\mathcal{D}$)
19: **function** EXPAND(n)
20: $succ = \emptyset$
21: **for all** $\alpha \in n.symbols$, such that $\alpha \in V$ **do**
22: Let $e_1 = n.symbols = u_1 \alpha u_2$
23: **for all** rules $r \in R, r : \alpha \rightarrow \beta$ **do**
24: Apply $e_1 \rightarrow_r e_2$ such that $e_2 = u_1 \beta u_2$
25: Create node ch and set $ch.symbols = e_2$
26: $succ = succ \cup \{ch\}$
27: **return** $succ$
28: **function** ESTIMATE(n, $\mathcal{D}$)
29: **for all** $i = 1$ to K **do**
30: Initialize $m = n$
31: **repeat**
32: $succ(m)$ = EXPAND(m)
33: Randomly select $ch \in succ(m)$, $m = ch$
34: **until** $\forall s \in m.symbols, s \in \Sigma$
35: Let $pipeline = m.symbols$
36: Set $s_i = \mathcal{H}(pipeline, \mathcal{D})$
37: **return** $average(s_1, \ldots, s_K)$

allow the pipeline data-flow graph to contain parallel data-processing paths followed by a concatenation of features (see the bottom of Figure 1(b) for a graphical representation of these recursive rules for symbol d). The reason behind the recursion ($d\&e$) is that an estimator is essentially a function – for example, binary classifiers generating class probabilities can be viewed as $e : \mathcal{D} \rightarrow [0, 1]$ – therefore can be used a new feature generator for the input dataset $\mathcal{D}$.

We say that a string $s \in \mathcal{L}(\mathcal{G}_{ML})$ containing only terminal symbols corresponds to a *terminal or complete pipeline structure*. Alternatively, a string $s \in (V \cup \Sigma)^*$ containing at least one non-terminal symbol represents a *partial pipeline structure*. Any partial pipeline can be extended to a complete one by applying the production rules defined by $\mathcal{G}_{ML}$. Furthermore, any pipeline structure $s \in (v \cup \Sigma)^*$ can be associated with a directed acyclic graph (DAG) $G = (U, E)$ where nodes correspond to symbols $\{e, t, d, N\}$ and there is a directed edge between any two nodes whose corresponding symbols are involved in a $>$ or $\&$ operators. For any sequence of consecutive $\&$ operators we introduce a *dummy node* that collects the edges from the nodes involved in the operators. In Figure 1(c) the dummy nodes are the solid black ones with three incoming edges. For clarity, we also add to G a *source node* that is connected to all nodes without incoming edges. By construction, G has only one *sink node*, namely a node without outgoing edges. Then, given a pipeline structure s and its corresponding DAG G, we define its length $l(s)$ as the longest path from the source to the sink node in G. The width of s, denoted $w(s)$ is defined as the largest in-degree of the dummy nodes of G. Clearly, if there are no dummy nodes in G then we assume $w(s) = 1$.

Example 1. *For illustration, in Figure 1(c) we show two possible complete pipeline structures that can be obtained using the grammar rules described above (we also include their corresponding strings). The first pipeline involves two parallel workflows consisting of a sequence of two transformations and a transformation followed by an estimation, respectively. The resulting new features are then concatenated with the original ones (which is facilitated by the middle NoOp step). This is then followed by another transformation and the final estimation step. The second pipeline starts with a transformation step followed by three parallel workflows involving additional transformations and estimations aimed at creating additional features which are then joined together and fed into the final estimation step. The pipeline structures have length 5 and width 3, respectively.*

Notice that, in contrast to other grammars proposed in the literature, such as (de Sa et al. 2017; Katz et al. 2020), our AutoML grammar does not commit the pipeline modules to specific transformers and estimators. Although it is possible to extend the grammar to include terminal symbols for transformers and estimators, we do not do that in this paper and focus instead on searching for the most promising shape.

Search for Pipeline Structures

The AutoML grammar $\mathcal{G}_{ML}$ defines a search space that allows us to search for the most promising DAG-shaped pipeline structure. Once such a structure is identified, we subsequently solve the combined algorithm selection and hyper-parameter optimization (CASH) problem (Thornton et al. 2012; Feurer et al. 2015) which selects the best machine learning methods for each of the pipeline steps and the corresponding hyper-parameters of these methods.

Clearly, the search space defined by $\mathcal{G}_{ML}$ is infinite. Therefore, in practice, we bound the length and width of the pipeline structures in order to traverse a finite search space. However, even with bounded length and width, the search space associated with DAG-shaped pipelines is much larger than that associated with linear pipelines. More specifically, given T transformers and E estimators, for a chain-shaped fixed pipeline of length l (width is 1), the number of possible pipelines is $O((T + E)^{l-1} \cdot E)$. For example, with $l = 4$, $T = 6$ and $E = 8$ we get over 21,000 pipelines not including hyper-parameters. In contrast, for DAG-shaped pipelines with maximum width w, the number of possible pipelines is bounded by $O((T + E)^{w \cdot (l-1)} \cdot E)$. With E, T as be-

fore and ($l = 6, w = 4$) (our setting used in experiments), the total possible pipelines without hyper-parameters is over 6.7×10^{23} which is clearly infeasible to search over without the grammar. This motivates our approach to first search for a promising pipeline structure and subsequently leverage any of the existing schemes for solving the CASH problem associated with the fixed pipeline structure identified.

Greedy Best-First Search

Algorithm 1 describes our greedy best-first search approach called PIPER (**PIP**eline gr**E**edy sea**R**ch) that finds and subsequently optimizes the most promising DAG-shaped pipeline for input dataset $\mathcal{D}$. In the following, $S_{\mathcal{G}}$ denotes the search space defined by the input AutoML grammar $\mathcal{G}_{ML} = \langle V, v_0, \Sigma, R\rangle$ where each node $n \in S_{\mathcal{G}}$ represents a partial pipeline structure and is associated with a string $n.symbols \in (V \cup \Sigma)^*$ that encodes the pipeline structure according to the grammar rules. Moreover, each node n is associated with a heuristic value (or h-value) denoted by $h(n)$. We use $succ(n)$ to denote n's successor nodes in $S_{\mathcal{G}}$. The list $OPEN$ maintains the search frontier and is assumed to be sorted in decreasing order of the nodes' h-values. Function EXPAND(n) describes how node n is expanded by generating its successors in the search space, while function ESTIMATE(n, $\mathcal{D}$) is used to compute n's h-value which estimates how promising the corresponding partial pipeline structure is for the input dataset $\mathcal{D}$. The root of $S_{\mathcal{G}}$ is labeled with the start symbol of $\mathcal{G}_{ML}$.

The search proceeds in the usual best-first manner. The node n with the best (i.e., highest) h-value is removed from the top of the $OPEN$ list and is expanded by generating its successors according to the grammar rules (line 13). Specifically, let $n.symbols$ be the string associated with the current node n. Then, for each non-terminal symbol $\alpha \in n.symbols$ such that $n.symbols = u_1 \alpha u_2$ and for each grammar rule $r \in R$ whose left hand side matches α (i.e., $r : \alpha \to \beta$) the algorithm creates a new child node labeled by the new string obtained from applying rule r on $n.symbols$. Furthermore, for each successor node $ch \in succ(n)$ we compute its corresponding h-value (line 16) before inserting it into the $OPEN$ list. If the current node n has no successors then its labeling string $n.symbols$ contains only terminal symbols and therefore it represents a complete pipeline structure. In this case, search terminates and PIPER subsequently optimizes the pipeline structure found by selecting the best configuration of transformers and estimators together with their hyper-parameters (line 22). This last stage is deferred to an off-the-shelf CASH optimizer as we will show next.

We note that during search PIPER can also cache symmetric pipeline structures (created due to ambiguity in the grammar) so that they are expanded only once.

Heuristic Function

As is common to best-first search algorithms, the effectiveness of PIPER depends on the guiding heuristic function. We show next how to compute the heuristic function $h(n)$ for each node n in the search space. Let n be the current search node and let $P = n.symbols$ be its corresponding (partial) pipeline structure. If P contains non-terminal symbols then we use the following strategy to compute $h(n)$. Let $\mathcal{S} = \{S_1, \ldots, S_K\}$ be a set of K randomly sampled terminal pipeline structures (in our experiments we used $K = 10$) that can originate from P (lines 37–42). For each pipeline S_i we use a surrogate model $\mathcal{H}$ to predict the accuracy of S_i on the current dataset $\mathcal{D}$, namely to compute $s_i = \mathcal{H}(S_i, \mathcal{D})$. Then $h(n)$ is the average accuracy over set $\mathcal{S}$ (alternatively, one can use the maximum accuracy over $\mathcal{S}$). If P is a terminal pipeline structure then $h(n) = \mathcal{H}(P, \mathcal{D})$. The surrogate model $\mathcal{H}$ is a random forest regressor trained on tuples of the form $((\mathbf{x}_k, \mathbf{p}_j), r)$ where $\mathbf{x}_k$ is vector of meta-features describing the k-th dataset $\mathcal{D}_k$, $\mathbf{p}_j$ is a vector of meta-features describing the j-th pipeline structure P_j and r is the predictive accuracy of P_j that was optimized for dataset $\mathcal{D}_k$ using an off-the-shelf optimizer. For our purpose, we considered 15 datasets from the OpenML repository (Vanschoren et al. 2013) and for each dataset we generated 100 random pipeline structures (based on grammar $\mathcal{G}_{ML}$) which we subsequently optimized using Hyperopt[1] (Bergstra et al. 2011) on a 70/30 split ratio for training and test data. The meta-features used to describe the datasets include standard statistical, information theoretic and landmark-based measures (Munoz et al. 2018), while the meta-features used to describe pipeline structures include the length and width of the pipeline structure, the number of estimators and transformers, as well as the number of $>$ and & symbols.

Optimizing a Pipeline Structure

As mentioned before, once PIPER finds a solution or a terminal pipeline structure it needs to solve the CASH problem associated with that particular structure for the input dataset (line 22). Although there have been many successful CASH optimizers developed over the past years (e.g., Auto-sklearn (Feurer et al. 2015) or MOSAIC (Rakotoarison, Schoenauer, and Sebag 2019)), these methods are typically restricted to linear pipeline structures and cannot handle directly our complex pipeline structures. Therefore, we extended a recent CASH optimizer called ADMM (Liu et al. 2020) to handle DAG-shaped pipeline structures as well.

The ADMM system (Liu et al. 2020) provides a framework for AutoML that formulates the problem of joint algorithm selection and hyper-parameter optimization as a formal mixed continuous-integer nonlinear program, and leverages the classic *alternating direction method of multipliers* to decompose this problem into two easier sub-problems: (i) black-box optimization with a small set of continuous variables (corresponding essentially to the continuous hyper-parameters), and (ii) a combinatorial optimization problem involving only Boolean variables (corresponding to the machine learning operators and discrete hyper-parameters). Consequently, these (potentially simpler) sub-problems can be effectively addressed by existing AutoML techniques, allowing ADMM to solve the overall problem effectively. For example, we can use Bayesian optimization (Snoek, Larochelle, and Adams 2012) to solve the continuous sub-problem and random search

[1] Available at https://pypi.org/project/hyperopt/. We extended Hyperopt to handle DAG-shaped pipelines.

(Bergstra and Bengio 2012) for the discrete optimization part, respectively.

Variants

We discuss next two variants of PIPER that extend the greedy best-first search beyond the first solution thus aiming at finding better pipeline structures. The first extension, called hereafter PIPER_X modifies Algorithm 1 as follows. For each leaf node $n \in S_{\mathcal{G}}$, namely $n.symbols$ encodes a terminal pipeline structure, PIPER_X computes a more informed estimate for $n.symbols$ than $h(n)$ by first sampling a random configuration for the transformers and estimators in $n.symbols$ together with their hyper-parameters, training the resulting pipeline on the training subset of the input dataset and subsequently evaluating the performance measure (e.g., classification error) on the test subset. This way, PIPER_X keeps track of the best possible pipeline structure until the search space is exhausted or a time limit is reached. Then, the pipeline structure corresponding to the best solution found is optimized for the input dataset using the same CASH optimizer as the original PIPER.

The second variant which we denote by PIPER_Z also continues the greedy best-first search after the first solution is found, but unlike PIPER_X, it optimizes the pipeline structure corresponding to each leaf node n using the same CASH optimizer as PIPER. In practice, however, PIPER_Z runs the CASH optimizer for a shorter amount of time compared with PIPER and PIPER_X, respectively.

Experiments

In this section, we evaluate empirically the performance of our proposed AutoML scheme for optimizing complex DAG-shaped pipelines. The algorithms were implemented in Python 3.6 using the scikit-learn algorithms (Pedregosa, Varoquaux, and Gramfort 2011) and we ran all experiments on a 2.6GHz CPU with 20GB of RAM.

Algorithms

We compare our proposed algorithms PIPER, PIPER_X and PIPER_Z with the TPOT system[2] (Olson et al. 2016) as well as two of the most recent and best-performing AutoML systems, called MOSAIC[3] (Rakotoarison, Schoenauer, and Sebag 2019) and ADMM (Liu et al. 2020), respectively. We do not compare with Auto-sklearn (Feurer et al. 2015) because both MOSAIC and ADMM outperformed it by a significant margin (Rakotoarison, Schoenauer, and Sebag 2019; Liu et al. 2020). ADMM was configured with combinatorial multi-arm bandits for solving the discrete optimization sub-problem and Bayesian optimization for the continuous one, using 25 iterations per sub-problem as suggested in (Liu et al. 2020). Furthermore, for TPOT we set the population size to 100. All competing systems except for MOSAIC used 6 transformers (MinMaxScaler, RobustScaler, StandardScaler, Normalizer, Nyströem, PCA) and 8 estimators (GaussianNB, GradientBoostingClassifier, KNeighborsClassifier, LogisticRegression MultinomialNB, LinearSVC, RandomForestClassifier, ExtraTreesClassifier). On the other hand, MOSAIC uses the search space defined in (Rakotoarison, Schoenauer, and Sebag 2019) which includes a much larger set of ML operators, namely 16 estimators and 21 transformers (consisting of 13 pre-processing methods, 6 rescaling strategies and 2 balancing strategies). Moreover, the current implementation of MOSAIC does not provide a mechanism for customizing the search space so that the user could easily choose the transformers and estimators to be used. While TPOT can evolve tree-shaped pipelines, ADMM and MOSAIC are restricted to a fixed linear pipeline having a pre-processing step, a transformation step followed by an estimation step. We didn't run RECIPE (de Sa et al. 2017) because its Python 2.7 implementation lead to package version mismatch and different implementation of the base operators thus preventing us from having a fair comparison with PIPER, TPOT, MOSAIC and ADMM which all use Python 3.6.

Benchmarks

We consider a collection of 50[4] binary and multi-class classification datasets from the OpenML repository (Vanschoren et al. 2013) containing missing values as well as both categorical and numerical features. To ensure a consistent evaluation, we converted the features of each dataset into numerical ones by first imputing any missing values with the most common value of the corresponding feature and subsequently one-hot encoding the categorical features. We report the *classification error* as the black-box objective (or performance measure) and evaluate it on a 70/30 train-test split for all competing algorithms. The total computational budget is set to 4 hours for each dataset and we average the performance over 10 independent runs. TPOT, MOSAIC, and ADMM use the entire time budget to optimize their corresponding fixed pipelines. PIPER and PIPER_X allocate the first 20 minutes (1200 seconds) to the greedy best-first search for finding the most promising DAG-shaped pipeline structure while the remaining time is used for optimizing the pipeline structure found. PIPER_Z allocates at most 10 minutes (600 seconds) for each terminal pipeline optimization and continues the search until the entire time budget is exhausted (see also the supplementary material).

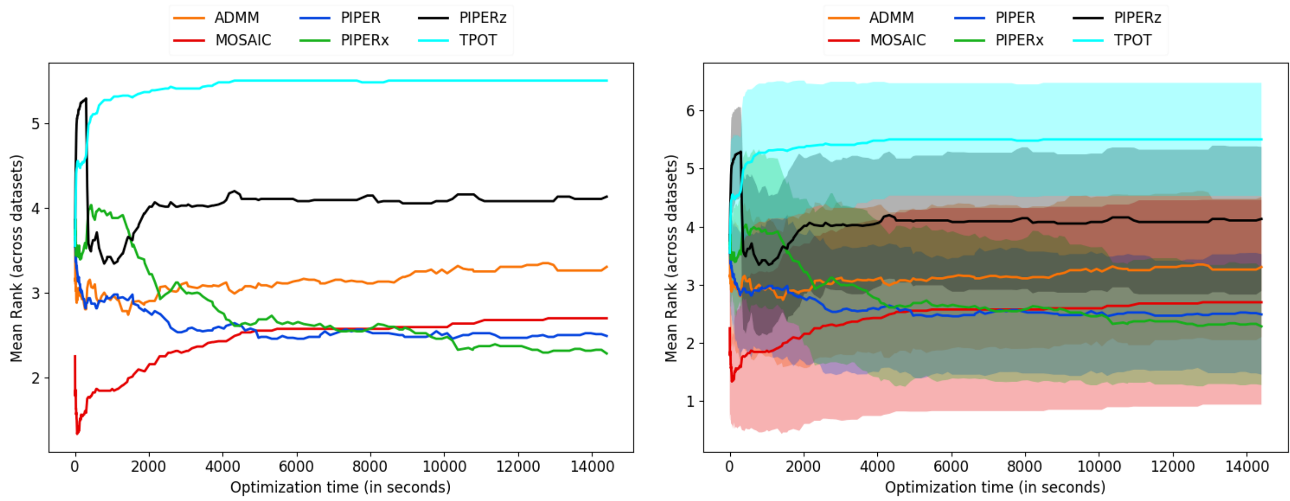
Results

Figure 2 plots the average ranks across all datasets of the mean performance over 10 runs (lower rank is better). We can see that MOSAIC is the best performing system achieving the best average rank within the first 1.5 hours. After that, both PIPER and PIPER_X catch up and steadily outrank MOSAIC with PIPER_X achieving the best rank after 4 hours, followed by PIPER and MOSAIC in the third place. TPOT is the worst performing system, while PIPER_Z and ADMM are clearly outperformed by MOSAIC, PIPER and PIPER_X. Note that in Figure 2(b), for the same set of operators (and

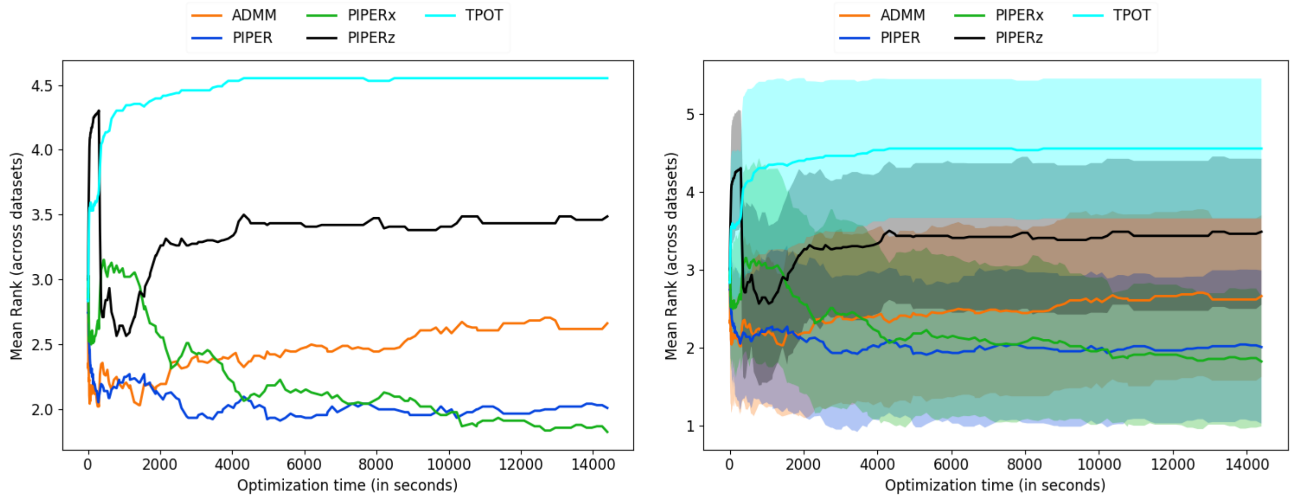
[2]Available at http://epistasislab.github.io/tpot/

[3]Available at https://github.com/herilalaina/mosaic_ml

[4]OpenML datasets: 3, 12, 15, 23, 29, 31, 36, 42, 54, 150, 179, 188, 469, 470, 991, 1053, 1067, 1169, 1461, 1464, 1468, 1475, 1486, 1489, 1492, 4135, 4532, 4538, 6332, 23381, 23517, 40685, 40981, 40996, 41002, 41138, 41142, 41143, 41146, 41147, 41150, 41159, 41161, 41163, 41164, 41165, 41166, 41167, 41168, 41169.



(a) All methods (mean rank and standard deviation)



(b) All methods except MOSAIC (mean rank and standard deviation)

Figure 2: Average rank (across 50 datasets) of mean performance across 10 runs (lower rank is better).

hyper-parameter ranges), PIPER and PIPER$_X$ searching over the pipeline structures show a significant performance improvement over existing fixed structured schemes as opposed to TPOT which is not competitive to fixed structured schemes for the 4 hour computation horizon. Figure 2(a) highlights that even with access to a smaller set of operators compared to MOSAIC, the pipeline structure search in PIPER and PIPER$_X$ is able to overcome this handicap to be competitive to MOSAIC, again demonstrating the value of structure search.

Figure 3 plots the mean performance (classification error) as a function of time, for 9 representative datasets (for clarity, we use log scale on both axes). First, we see that MOSAIC finds reasonably good pipelines relatively quickly which is primarily due to its effective Monte-Carlo tree search strategy but also to the extended set of transformers and estimators used. PIPER typically finds the most promising pipeline structure before the 20 minute mark and spends the remaining time optimizing it which in many cases converges to a good performance value. PIPER$_X$ is slower to find the best pipeline structure but because it explores a larger space than PIPER in many cases is able to optimize a better structure which often translates into better performance. Both PIPER and PIPER$_X$ are very competitive with ADMM and MOSAIC, in many cases converging to much better objective values. In contrast, PIPER$_Z$ is very slow in converging to good results and this is because it only allocates a relatively small amount of time (10 minutes) for optimizing the pipeline structures visited. Finally, TPOT is the worst performing system across the majority of datasets considered.

In Table 1 we give the pipeline structures (together with the classification errors) that were found and subsequently

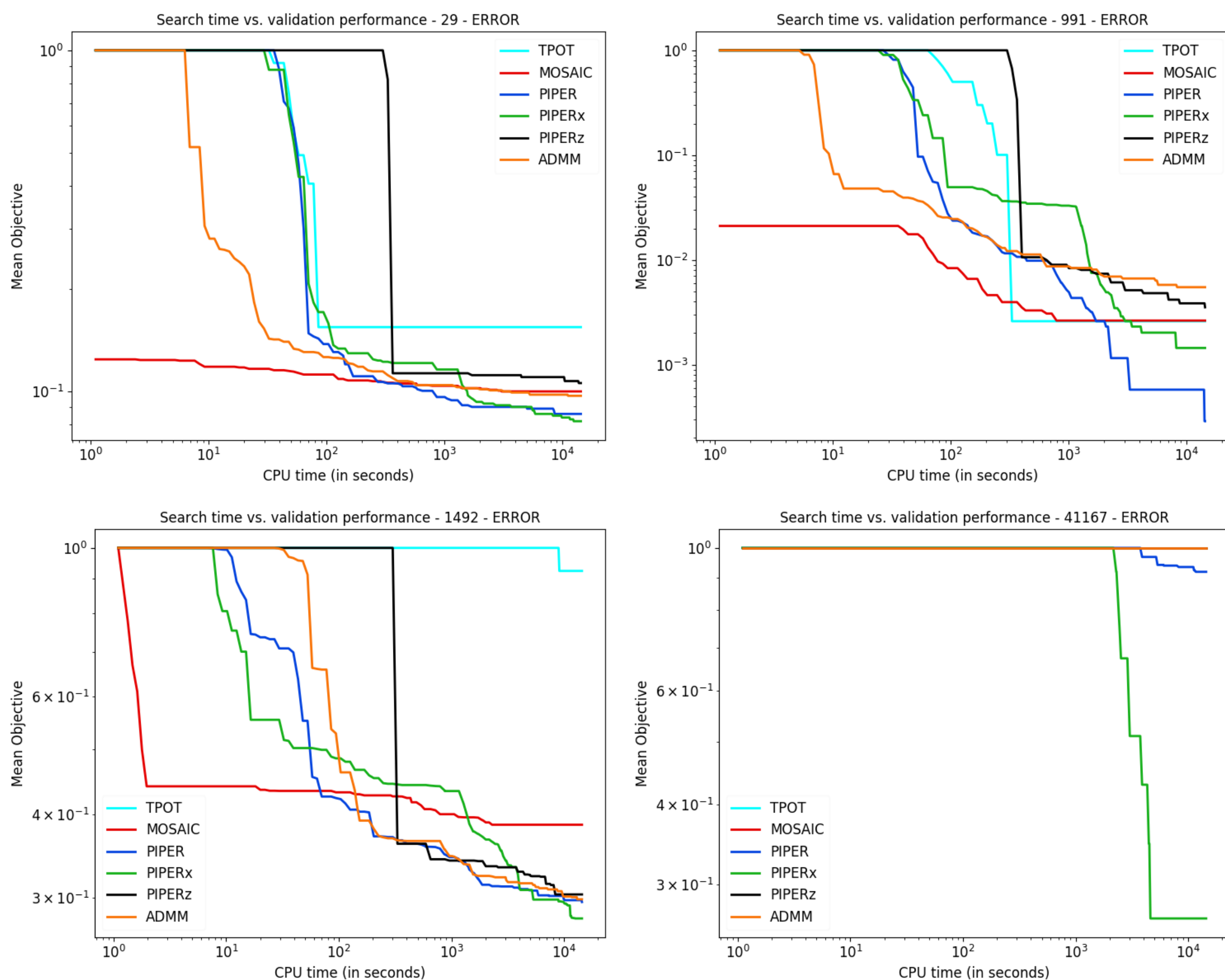


Figure 3: Mean performance (across 10 runs) vs time (in seconds) for representative datasets.

dataset	PIPER$_X$	error
29	((t>((t&e)&e))&(t>(t&e)))>e	0.0818
991	(((t>t>t)>((t&t)>t))&e)>e	0.0014
1492	(t>(t&e))>e	0.2797
41167	(t&t)>e	0.2658

Table 1: Pipeline structures optimized by PIPER$_X$.

optimized by PIPER$_X$ on the datasets from Figure 3. We observe a wide variety of shapes that in many cases lead to improved performance over the competitors.

Conclusion

The paper explores the power of heuristic search and context-free grammars to generate machine learning pipelines having directed acyclic graph shapes. These more complex pipelines go beyond the traditional fixed linear workflows and have the potential to discover additional hidden features which in turn could translate into improved performance. We developed a greedy best-first search scheme that first traverses the search space of possible pipeline structures in order to identify the most promising one and subsequently optimizes the best structure found by selecting the best configuration of machine learning algorithms and their hyper-parameters. We also discuss several variants of this approach. Our empirical evaluation on a variety of datasets demonstrates the competitiveness of our approach which often outperforms current state-of-the-art AutoML systems. Future work includes the development of new heuristic search algorithms, more accurate heuristic functions as well as interleaved computation between heuristic search and pipeline optimization.

References

Assuncao, F.; Lourenco, N.; Ribeiro, B.; and Machado, P. 2020. Evolution of Scikit-Learn Pipelines with Dynamic Structured Grammatical Evolution. In *International Con-*

ference on the Application of Evolutionary Computation, 530–545.

Bergstra, J.; and Bengio, Y. 2012. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research* 13(1): 281–305.

Bergstra, J. S.; Bardenet, R.; Bengio, Y.; and Kegl, B. 2011. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems (NIPS)*, 2546–2554.

Chen, B.; Wu, H.; Mo, W.; Chattopadhyay, I.; and Lipson, H. 2018. Autostacker: A compositional evolutionary learning system. In *Genetic and Evolutionary Computation Conference (GECO)*, 402–409.

de Sa, A.; Freitas, A.; and Pappa, G. 2018. Automated Selection and Configuration of Multi-Label Classification Algorithms with Grammar-Based Genetic Programming. In *International Conference on Parallel Problem Solving from Nature*, 308–320.

de Sa, A.; Pinto, W.; Oliveira, L.; and Pappa, G. 2017. RECIPE: A Grammar-Based Framework for Automatically Evolving Classification Pipelines. In *European Conference on Genetic Programming (EuroGP)*, 246–261.

Dechter, R.; and Pearl, J. 1985. Generalized Best-First Search Strategies and the Optimality of A*. *In Journal of ACM* 32(3): 505–536.

Drori, I.; Krishnamurthy, Y.; Rampin, R.; de Paula Lourenco, R.; Ono, J.; Cho, K.; Silva, C.; and Freire, J. 2018. Alphad3m: Machine learning pipeline synthesis. In *Workshop on AutoML (ICML)*.

Feurer, M.; Klein, A.; Eggensperger, K.; Springenberg, J.; Blum, M.; and Hutter, F. 2015. Efficient and robust automated machine learning. In *Advances in Neural Information Processing Systems*, 2962–2970.

Guyon, I.; Bennett, K.; and Cawley, G. 2015. Design of the 2015 Chalearn AutoML challenge. In *IJCNN*, 1–8.

Guyon, I.; and Tu, W.-W. 2018. Automatic Machine Learning Challenge 2018: Towards AI for Everyone. In *PAKDD 2018 Data Competition*.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research* 26: 191–246.

Hirzel, M.; Kate, K.; Shinnar, A.; Roy, S.; and Ram, P. 2019. Type-Driven Automated Learning with Lale. *CoRR* abs/1906.03957. URL http://arxiv.org/abs/1906.03957.

Hutter, F.; Hoos, H.; and Leyton-Brown, K. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization (LION)*, 507–523.

Kandasamy, K.; Neiswanger, W.; Schneider, J.; Poczos, B.; and Xing, E. 2018. Neural architecture search with Bayesian optimisation and optimal transport. In *NeurIPS*.

Katz, M.; Ram, P.; Sohrabi, S.; and Udrea, O. 2020. Exploring Context-Free Languages via Planning: The Case for Automating Machine Learning. In *International Conference on Automated Planning and Scheduling*, 403–411.

Kotthoff, L.; Thornton, C.; Hoos, H.; Hutter, F.; and Leyton-Brown, K. 2017. Auto-weka 2.0: Automatic model selection and hyperparameter optimization in weka. *J. Mach. Learn. Res.* 18(1): 826–830.

Liu, H.; Simonyan, K.; Vinyals, O.; Fernando, C.; and Kavukcuoglu, K. 2018. Hierarchical representations for efficient architecture search. In *ICLR*.

Liu, H.; Simonyan, K.; and Yang, Y. 2019. DARTS: Differentiable Architecture Search. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

Liu, S.; Ram, P.; Vijaykeerthy, D.; Bouneffouf, D.; Bramble, G.; Samulowitz, H.; Wang, D.; Conn, A.; and Gray, A. 2020. An ADMM Based Framework for AutoML Pipeline Configuration. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

Mohr, F.; Wever, M.; and Hullermeier, E. 2018. ML-Plan: Automated machine learning via hierarchical planning. *Machine Learning* 107(1): 1495–1515.

Munoz, M. A.; Villanova, L.; Baatar, D.; and Smith-Miles, K. 2018. Instance spaces for machine learning classification. *Machine Learning* 107(1): 109–147.

Nilsson, N. J. 1980. *Principles of Artificial Intelligence.* Tioga.

Olson, R.; Bartley, N.; Urbanowicz, R.; and Moore, J. 2016. Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science. In *Genetic and Evolutionary Computation Conference (GECO)*, 485–492.

P. Hart, N. N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 8(2): 100–107.

Pearl, J. 1984. Heuristics: Intelligent Search Strategies. In *Addison-Wesley*.

Pedregosa, F.; Varoquaux, G.; and Gramfort, A. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12(1): 2825–2830.

Pham, H.; Guan, M. Y.; Zoph, B.; Le, Q. V.; and Dean, J. 2018. Efficient Neural Architecture Search via Parameter Sharing. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, 4092–4101.

Rakotoarison, H.; Schoenauer, M.; and Sebag, M. 2019. Automated Machine Learning with Monte-Carlo Tree Search. In *International Joint Conference on Artificial Intelligence (IJCAI)*.

Real, E.; Aggarwal, A.; Huang, Y.; ; and Le, Q. 2019. Regularized evolution for image classifier architecture search. In *AAAI*.

Segovia-Aguas, J.; Jimenez, S.; and Jonsson, A. 2017. Generating context-free grammars using classical planning. In *International Joint Conference on Artificial Intelligence (IJCAI)*, 4391–4397.

Snoek, J.; Larochelle, H.; and Adams, R. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems (NIPS)*, 2951–2959.

Thornton, C.; Hoos, H.; Hutter, F.; and Leyton-Brown, K. 2012. Auto-weka: Automated selection and hyper-parameter optimization of classification algorithms. In *CoRR, abs/1208.3719*.

Vanschoren, J.; van Rijn, J.; Bischl, B.; and Torgo, L. 2013. OpenML: Networked Science in Machine Learning. *SIGKDD Expl.* 15(2): 49–60.

Wang, L.; Zhao, Y.; Jinnai, Y.; Tian, Y.; and Fonseca, R. 2019. AlphaX: eXploring Neural Architectures with Deep Neural Networks and Monte Carlo Tree Search. In *arXiv:1903.11059*.

Wever, M.; Mohr, F.; and Hullermeier, E. 2018. ML-Plan for Unlimited-Length Machine Learning Pipelines. In *AutoML Workshop (ICML)*.

Zoph, B.; and Le, Q. 2017. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*.